Introduction à la programmation fonctionnelle

Insalgo

2024

Lambda Calcul

Contexte historique

Dans les années 1930, plusieurs mathématiciens étudient la théorie de la calculabilité : ce sont les débuts de l'informatique. Cette théorie étudie les problèmes avec des solution **algorithmiques**. Mais à l'époque, il n'y a pas de langages pour décrire les algorithmes. Deux mathématiciens proposent une manière de décrire les algorithmes :

- Alan Turing propose la **machine de Turing**
- Alonzo Church propose le λ-**calul**

Il sera démontré que ces deux paradigmes bien que très différents dans leur approche sont équivalents (la notion de Turing-complétude a été inventée pour cette démonstration)

Definition du lambda calcul

Le lambda calcul est défini de manière très simple : on construit un **terme** (un programme) en composant 3 blocs de base :

- Les variables x
- Les abstractions $\lambda x.M$ où M est un terme (= déclaration de fonction)
- Les applications $M\ N$ où M et N sont des termes (= appel de fonction)

Sur lesquels on peut appliquer deux opérations logiques :

- α-conversion : $\lambda x.M[x] \rightarrow \lambda y.M[y]$ (changement de variable)
- β-reduction : $(\lambda x.M[x])\ N \rightarrow M[x := N]$ (application de la fonction)

Équivalents lambda calcul/python

Le terme $\lambda x.x$ s'écrit

```
lambda x: x
```

Le terme $(\lambda x.x)y$ s'écrit

```
(lambda x:x)(y)
```

Le terme $\lambda x.f\ x$ s'écrit

```
lambda x:f(x)
```

Exemples de calculs

$(\lambda x.x)\ y \rightarrow y$
$(\lambda x.f\ x)y \rightarrow f\ y$

Définition fonctionnel

Le paradigme fonctionnel

Le paradigme de la programmation fonctionnelle provient du λ-calcul. On retrouve donc les concepts suivants :

- Fonctions pures
- Variables immutables
- Fonctions "first-class citizen"

Mais il y a aussi beaucoup d'ajouts par rapport à ce modèle théorique très simple (liste non exhaustive):

- Manipuler autre chose que des fonctions (strings, nombres, booléens,...)
- Algebraic Data Types
- Pattern matching
- Système de types avancé (pour certains langages)

Fonctions pures

Une fonction pure est l'analogue informatique des fonction mathématiques. Une fonction pure est une fonctions qui valide plusieurs critères :

- Pas d'effets de bord : pas de mutation de variables globales ou d'entrées mutables
- Déterministes : les mêmes paramètres donne le même résultat

Fonctions pures

Une fonction pure est l'analogue informatique des fonction mathématiques. Une fonction pure est une fonctions qui valide plusieurs critères :

- Pas d'effets de bord : pas de mutation de variables globales ou d'entrées mutables
- Déterministes : les mêmes paramètres donne le même résultat

Avantages

- Reproductible et testable !
- Se parallélise facilement
- Facilité de raisonnement sur le code

Concepts

Vous avez dit boucles ?

Non !
Une boucle implique une variable mutable. Or, les variables doivent être immutables.
Pas de boucles `while` ou `for`.

Vous avez dit boucles ?

Non !
Une boucle implique une variable mutable. Or, les variables doivent être immutables.
Pas de boucles `while` ou `for`.

Solution

Les boucles `for` peuvent souvent être remplacées par des **fonctions d'ordre supérieur**. De manière générale, les boucles sont remplacées par de la **récursion**. Les langages fonctionnels ont des optimisation pour rendre la récursion plus performante (tail recursion).

Fonctions "first-class citizen"

On dit que les fonctions sont des citoyens de première classe dans un langage quand celui-ci permet de traiter les fonctions comme des valeurs : les assigner à des variables, les passer en paramètres à d'autres fonctions.
Par exemple, en python, les fonctions sont bien des citoyens de première classe, il est possible de les assigner à des variables et de les passer en paramètre :

```
def application(f, x):
    return f(x)

def ajouter_un(x):
    return x+1

resultat = application(ajouter_un, 3) # 4
```

Fonctions d'ordre supérieur

Une fonction d'ordre supérieur est une fonctions qui prend au moins une fonction en paramètre et/ou retourne une fonctions.
Python fournit quelques de fonctions d'ordre supérieur, an particulier pour le traitement des listes :

```
res = map(ajouter_un, [1, 2, 3, 4]) # [2, 3, 4, 5]
res2 = filter(res, lambda x: x>2) # [3, 4, 5]
```

Le paquet functools fournit plus de fonctions d'ordre supérieur.

```
res = reduce(lambda x, y: x+y, [1,2,3,4]) # 10
```

Composition de fonctions

La programmation fonctionnelle se repose beaucoup sur le faire de faire le bon enchaînement de fonctions pour parvenir jusqu'au résultat attendu. Lorsqu'on enchaîne deux fonctions, on dit qu'on les compose : $\forall x, g(f(x)) = (g \circ f)(x)$ Grâce aux fonctions d'ordre supérieur, on peut appliquer le même raisonnement en programmation.
Une idée d'implémentation ?

Composition de fonctions

La programmation fonctionnelle se repose beaucoup sur le faire de faire le bon enchaînement de fonctions pour parvenir jusqu'au résultat attendu. Lorsqu'on enchaîne deux fonctions, on dit qu'on les compose : $\forall x, g(f(x)) = (g \circ f)(x)$ Grâce aux fonctions d'ordre supérieur, on peut appliquer le même raisonnement en programmation.

```
def compose(g, f):
    return lambda x : g(f(x))

add = lambda x: x+1
mult = lambda x: x*2
add_then_mult = compose(mult, add)
res = add_then_mult(2) # = (2+1)*2 = 6
```

Applications partielles et currying

Définition

On dit qu'on **applique partiellement** une fonction lorsqu'on ne donne qu'une partie de des paramètres, on obtient alors une nouvelle fonction qui attend les paramètres restants. Soit f une fonction de $\mathbb{R}^2$ dans $\mathbb{R}$ tq. $\forall (x, y) \in \mathbb{R}^2, f(x, y) = x + y$ alors la fonction g de $\mathbb{R}$ dans $\mathbb{R}$ tq. $\forall x \in \mathbb{R}, g(x) = f(2, x)$ est une application partielle de f. Lorsqu'on programme, on peut aussi avoir à un moment donné une partie des arguments pour une fonction, mais ne pas avoir les autres. Il est alors utile d'obtenir une application partielle de la fonction avec les arguments déjà connus.

Applications partielles et currying

Solutions sans currying

Il y a plusieurs moyens d'arriver au même résultat en python :
Une fonction g comme dans la définition :

```
def f(x, y):
    return x+y
def g(x):
    return f(2,x)
res = g(5) # 2+5 = 7
```

Ou bien l'utilisation de la fonction partial de functools

```
def f(x, y);
    return x+y
g = partial(f, 2)
res = g(5) # 2 + 5 = 7
```

Applications partielles et currying

Currying

Définition

Revenons à notre λ-calcul. Dans la grammaire du langage, seules les fonctions à une variable d'entrée sont autorisées, comment alors représenter notre fonction une fonction qui prend deux arguments ?

Applications partielles et currying

Currying

Définition

Revenons à notre λ-calcul. Dans la grammaire du langage, seules les fonctions à une variable d'entrée sont autorisées, comment alors représenter notre fonction une fonction qui prend deux arguments ?
Comme cela : $\lambda x.\lambda y.M[x, y]$ (notation raccourcie : $\lambda xy.M[x, y]$)
Ce mécanisme se nomme currying (du nom de Haskell Curry qui l'a inventé)

Explication

Une idée ? Tentons de réduire $(\lambda x.\lambda y.M[x, y])\ N\ P$

Applications partielles et currying

Currying

Définition

Revenons à notre λ-calcul. Dans la grammaire du langage, seules les fonctions à une variable d'entrée sont autorisées, comment alors représenter notre fonction une fonction qui prend deux arguments ?
Comme cela : $\lambda x.\lambda y.M[x, y]$ (notation raccourcie : $\lambda xy.M[x, y]$)
Ce mécanisme se nomme currying (du nom de Haskell Curry qui l'a inventé)

Explication

$(\lambda x.\lambda y.M[x, y])\ N\ P$
$\rightarrow (\lambda y.M[x := N, y])\ P$
$\rightarrow M[x := N, y := P]$

Applications partielles et currying

Application du currying aux application partielles

On constate à l'étape intermédiaire que le terme $\lambda y.M[x := N, y]$ est l'application partielle avec N comme premier argument de notre fonction. Le currying nous offre gratuitement l'application partielle comme étape intermédiaire.
Ainsi, $(\lambda x.\lambda y.M[x, y])\ N$ la fonction prend deux paramètres mais si on ne lui en fournit qu'un seul paramètre N, le terme vaut l'application partielle de la fonction.
Une idée d'implémentation en Python ?

Applications partielles et currying

Application du currying aux application partielles en python

On peut coder ce mécanisme en python, bien que le langage n'y soit pas très adapté.

```
def add(a):
    def partial(b) :
        return a+b
    return partial

partial = add(5)
res1 = partial(2) # 7
res = add(2)(3) # 5
```

On peut définir add uniquement en fonction lambda (l'utilisation est la même)

```
add = lambda a: lambda b: a+b
```

Algebraic Data Types

Types multiplicatifs

Les **types algébriques de données** sont des types composés. C'est-à-dire qu'on crée un nouveau type à partir d'autre types existants.
L'exemple le plus connu est une struct ou classe :

```
class A :
    a: B
    b: C
```

Ces types sont dits **multiplicatifs** car ils sont l'équivalent de l'opération mathématique du **produit cartésien**. On peut en effet représenter le type comme cela : $A = B \times C$.
Ou aussi car $|A| = |B| \times |C|$
En effet, si B a n valeurs possibles et C a m valeurs possibles alors A a $m \times n$ valeurs possibles.

Algebraic Data Types

Types additifs

L'autre manière de composer des types est par des types additifs :

```
type A = B | C
```

Une variable de type A est en fait de type B OU de type C. On dit que ces types sont additifs car ils sont équivalents à l'union disjointe $A = B + C$ et aussi car $|A| = |B| + |C|$ En effet, si B a n valeurs possibles et C a m valeurs possibles alors A a $m + n$ valeurs possibles.
Ces types additifs sont aussi appelés **unions**.

Algebraic Data Types

Unions taggées

L'union en Python est non taggée, c'est à dire qu'on ne peut pas identifier facilement de quelle variante est une variable
Certains langages proposent des **unions taggées**, c'est à dire que chaque variante a un identificateur appelé tag.

- En Rust :

```
enum Exemple {
    Variante1(i64),
    Variante2(f64)
}
```

- En Haskell :

```
data Pair = Variante1 Int
          | Variante2 Double
```

Pattern matching

Variantes

Lorsque l'on manipule un type additif, on doit pouvoir faire une actions différente selon la variante, et en extraire les données internes. Ce là qu'intervient le pattern matching. En Rust :

```
let exemple = Exemple::Variante2(5.3);
let res = match exemple {
    Exemple::Variante1(inner) => inner + 1,
    Exemple::Variante2(inner) => inner as i64 + 2
}; // res = 7
```

Ce mécanisme nous permet de distinguer entre les variantes et d'extraire les données contenues dans chaque variante.

Pattern matching

En python

Python n'a pas d'union taggées alors il faut créer des classes et le nom de ces classes servent de tag.

```
@dataclass
class B:
    x: int
@dataclass
class C:
    x: float
    y: bool
type A = B | C

exemple: A = C(3.5, True)
match exemple:
    case B(a): print(f"B contains {a}")
    case C(a, b): print(f"C contains {a} and {b}")
```

Pattern matching

En python

En réalité le pattern matching est bien plus puissant que ce qui a été présenté. Il permet de détecter la structure des données et de faire différentes actions selon la structure, en particulier des listes et tuples.

```
match command.split():
    case ["quit"]: ...
    case ["go", dir] | ["go", "to", dir]: ...
    case ["turn", ("right"|"left") as dir] : ...
    case ["say", word] if word not in banned : ...
    case ["drop", *objects]: ...
    case _:
        print(f"Commande inconnue")
```

Voir PEP 636 pour un tour complet des fonctionnalités.

Gestion d'erreurs et valeurs nulles

Le problème

Nous avons vu qu'en programmation fonctionnelle, les fonctions doivent avoir une valeur de retour pour toutes les combinaisons possibles de valeurs d'entrée. Mais que faire quand les valeurs d'entrée sont invalides ? Par exemple, la division n'est pas valide pour toutes les combinaisons de deux nombres, si le dénominateur vaut 0, alors la division n'est pas possible.
Comment alors créer une fonction `div(x: float, y: float)` qui reste pure ?

Gestion d'erreurs et valeurs nulles

Le type Maybe / Option

Il faut modéliser dans le type de retour le fait que la fonction peut échouer. Cela se fait par le type `Maybe` (en Haskell) / `Option` (en Rust). Voici une implémentation de ce type en Python (version 3.12 minimum):

```
@dataclass
class Just[T] :
    val: T
class Nothing:
    ...
type Maybe[T] = Just[T] | Nothing
```

`Maybe` / `Just` / `Nothing` sont les noms en Haskell. Les noms équivalents en rust sont `Option` / `Some` / `None`

Une idée d'implémentation pour div(x: float, y: float) ?

Gestion d'erreurs et valeurs nulles

Exemple d'implémentation

On peut implémenter la fonction de cette manière

```
def div(a: float, b: float) -> Maybe[float]:
    if b == 0 :
        return Nothing()
    return Just(a/b)
```

Puis on peut faire des opérations sur le résultat avec `match` :

```
match div(3, 5):
    case Just(res): print(f"Le resultat est {res}")
    case Nothing(): print("Division interdite !!")
```

Gestion d'erreurs et valeurs nulles

Attention à ne pas confondre `Nothing` et `None`, `Nothing()` est une valeur valide seulement pour une variable de type `Maybe`, alors que `None` représente l'absence de valeur pour tous les types.
En Rust et Haskell, le mot clé `none`/`null`/`nil` est totalement absent. Pour représenter l'absence de valeur, il est nécéssaire de donner le type `Option`/`Maybe` et de donner sa variante `Nothing` (appelée `None` en Rust)

Gestion d'erreurs et valeurs nulles

Avantages

Ainsi, on modélise les erreurs dans le système de types, les erreurs sont traitées comme de simples valeurs. Il n'y a pas de mécanismes spécialisés pour la gestion d'erreurs (comme les exceptions) Cela permet de savoir si une fonction peut échouer simplement en regardant sa signature, et de s'assurer de gérer l'erreur.
Lorsqu'on veut chainer des opérations qui peuvent échouer, la syntaxe peut rapidement devenir chargée. Les langages type Rust ou Haskell qui utilisent par défaut cette gestion d'erreurs apportent des mécanismes qui permettent de simplifier cela (opérateur ? en Rust, opérateurs >>= et>> en Haskell).
En Haskell, ces opérateurs sont plus génériques et s'appliquent en réalité à tout un ensemble de types qui sont des **Monades**, mais pas besoin d'en savoir plus pour les utiliser.

Conclusion

Conclusion

Les programmation fonctionnelle peut être déroutante au début. En particulier les langages fonctionnels ont une syntaxe particulière et de nouveaux concepts à comprendre.
On peut tout de même reprendre certains concepts dans des langages impératifs plus classiques.
Le programmation fonctionnelle peut permettre un style plus impératif permettant de mieux décrire les intentions, us système de types plus puissant qu'habituellement et une nouvelle visions sur les problèmes posés.